New Results on Server Problems

M. Chrobak * H. Karloff † T. Payne ‡ S. Vishwanathan §

Abstract

In the *k-server problem*, we must choose how k mobile servers will serve each of a sequence of requests, making our decisions in an online manner. We exhibit an optimal deterministic online strategy when the requests fall on the real line. For the *weighted-cache problem*, in which the cost of moving to x from any other point is $w(x)$, the weight of x, we also provide an optimal deterministic algorithm. We prove the nonexistence of competitive algorithms for the asymmetric two-server problem, and of memoryless algorithms for the weighted-cache problem. We give a fast algorithm for offline computing of an optimal schedule, and show that finding an optimal offline schedule is at least as hard as the assignment problem.

1 Introduction

The k-server problem can be stated as follows. We are given a metric space M, and k *servers* which move among the points of M, each occupying one point of M. Repeatedly, a *request* (a point $x \in M$) appears. To *serve* x, each server moves some distance, possibly 0, after which the point x must be occupied by one of our servers. The *cost* incurred is the sum of the k distances moved. We must serve this request by considering only the current and past requests: the decisions are made *online*. The server problem encompasses many interesting problems as special cases, for example: heuristics for linear search [14], paging [9, 13], font caching in printers [13] and motion planning for 2-headed disks [4].

Let us call a sequence of k not-necessarily-distinct points of a metric space a *configuration*. If M is a metric space, we call an online strategy $\mathcal{S}$ c_k*-competitive for* M if for

*Department of Mathematics and Computer Science, University of California, Riverside, CA 92521.

†Department of Computer Science, University of Chicago, Chicago, IL 60637. Research supported by the NSF grant CCR 8807534.

‡Department of Mathematics and Computer Science, University of California, Riverside, CA 92521.

§Department of Computer Science, University of Chicago, Chicago, IL 60637.

every initial configuration $R = (r_1, r_2, ..., r_k)$, there is a real a such that the following property holds. Let σ be an arbitrary request sequence for M and let $OPT_R(\sigma)$ be the optimal (offline) cost to serve σ, when initially the ith server occupies r_i. Then the total cost incurred by $\mathcal{S}$ on σ, when its ith server starts in r_i, is at most

$$c_k\, OPT_R(\sigma) + a.$$

Here, the benchmark is the minimum cost needed to serve the request sequence, minimized over all possible ways of serving this sequence. Thus for $\mathcal{S}$ to be c_k-competitive, its cost must not exceed a more than c_k times the cost of the optimal offline algorithm. We say $\mathcal{S}$ is *competitive for M* if it is c_k-competitive for M, for some c_k. Strategy $\mathcal{S}$ is *c_k-competitive* or *competitive* if the respective definitions hold for all metric spaces. It is known [10] that no c_k-competitive strategy exists if $c_k < k$. Also, no generality is lost in assuming, if desired, that the initial locations r_i are distinct.

Yet a more general model, that of *task systems*, was studied by Borodin, Linial and Saks [2], who gave an optimal online algorithm in their model. However, in their approach the "competitive ratio" is allowed to depend on the cardinality of the metric space. Our problems are less general but sometimes permit stronger results.

At the moment, for no $k \geq 3$ is any competitive (deterministic) algorithm known for all metric spaces. Several important results were obtained by Manasse, McGeoch and Sleator [10]. They presented a 2-competitive algorithm for the 2-server problem and an $(n-1)$-competitive algorithm for the $(n-1)$-server problem on n-point metric spaces. It was also Manasse *et al.* who showed that for any metric space with $k+1$ or more points, no c_k-competitive strategy exists for that metric space if $c_k < k$. Recently, Irani and Rubinfeld [8] proved that a version of a balancing algorithm is 10-competitive for two servers.

One approach to the problem is to seek a randomized strategy for which the expected value of the cost does not exceed a plus c_k times the optimal cost. The lower bound mentioned above collapses in the randomized model. In fact, for the *paging* problem—the allowable metric spaces are those with all unit distances—Fiat *et al.* [7] presented a strategy for paging that is $2H_k$-competitive, where H_k is the kth harmonic number. (H_k is asymptotic to $\ln k$.) They also proved that for the paging problem no strategy can be c_k-competitive unless $c_k \geq H_k$. Thus the H_k-competitive algorithm presented by McGeoch and Sleator [12] is optimal.

In Section 2 we consider the case of k servers on a line. The simple algorithm we present is k-competitive and hence optimal. This is the first competitive deterministic algorithm for k servers in a metric space with unboundedly large distances. Our algorithm in addition is *memoryless* in that the algorithm can be specified by a function $f : M^{k+1} \to M^k$. When in configuration $(r_1, r_2, ..., r_k)$ with a request at Q, the servers move to $f(r_1, r_2, ..., r_k, Q) = (p'_1, p'_2, ..., p'_k)$, the ith moving from r_i to p'_i (of course $Q \in \{p'_1, ..., p'_k\}$). Such an algorithm keeps no record of the past other than the configuration itself. The importance of memoryless algorithms has been emphasized by Raghavan and Snir [13]. (One should be aware of the possibility that in certain infinite metrics space one might be able to store the entire history of the computation in the location of one or more servers.)

Section 3 proves that a simple balancing algorithm is k-competitive—and thus optimal—for the *weighted-cache problem*, in which associated with each point x is a positive weight $w(x)$ and the distance to x from any other point is $w(x)$. Thus, we allow the "metric" to be asymmetric and apply the definitions above. (We say an *asymmetric metric space* is a space for which $d(x, y)$ is nonnegative and $d(x, y) = 0$ implies $x = y$. Distances must obey the triangle inequality yet need not be symmetric.) This problem was proposed by Manasse *et al.* [11], and a randomized k-competitive algorithm was given by Raghavan and Snir [13]. In the same section we also present a new memoryless algorithm for the unweighted case.

In Section 4 we first show that for the general asymmetric problem, there can be no competitive algorithm, even for two servers. This stands in contrast to the conjecture "that among all k-server problems, the ones that have the highest competitive factor are the symmetric ones," made by Manasse *et al.* in [11]. We then show that there is no deterministic memoryless algorithm for the weighted-cache problem.

In Section 5 we consider offline algorithms for the server problem. We give a $O(kn^2)$-time algorithm to find an optimal schedule for k-servers and n requests. Following this result is a proof that when $k = n/2$ this problem is at least as hard as the assignment problem on n-node bipartite graphs. Last, we show that on the line (where the input size is n), the time needed to find an optimal schedule in the algebraic computation tree model is $\Omega(n \log n)$.

By virtue of the triangle inequality, an optimal strategy can be assumed never to move more than one server while serving a request; see [10]. However, with concurrent motion of the servers the algorithm for the line is memoryless and the proofs are simpler. When we simulate our algorithm with one that moves only one server at a time, this new algorithm will need memory to store the virtual positions of our servers.

Often, our goal will be to show that an online algorithm we construct is competitive. To do this, we will conceptually create an *adversary* who, with his own k servers, must serve the same sequence of requests as we serve, starting from the same configuration, but who knows—indeed, who chooses—the entire request sequence in advance. If we can prove that our cost is no more than a plus k times the adversary's cost on the same request sequence, despite the adversary's foresight, then our online algorithm must be k-competitive.

2 A Memoryless Algorithm for the Line

In this section we present an algorithm for k servers on a line. The algorithm is simple, memoryless, and achieves the optimal ratio: k. Each request is specified by a real number, the location on the line of the request. Here is the algorithm.

Algorithm DOUBLE-COVERAGE: Where s denotes our server closest to the request P, and d is the distance from s to P, serve the request with s. Then, if we have any servers on the "side" of P opposite to s (e.g., to P's left if s is to P's right), move the closest one d units toward P. Thus, we actually move one or two servers in response to a request, moving one if and only if all of our servers are on the same side of P.

We imagine that in response to a request, first the adversary serves the request, all of our servers remaining stationary. After the adversary has served the request he stays stationary while we serve the request. S^* and A^* denote the total cost we and the adversary incur over the whole sequence of requests, respectively. We label our k servers $s_1, ..., s_k$; the adversary's k servers are labeled $x_1, ..., x_k$. We will also use s_i and x_j to denote real numbers giving the current locations of servers s_i and x_j on the real line. By relabeling the servers as they move along the line, if necessary, we may assume that always $s_1 \leq s_2 \leq \cdots \leq s_k$ and $x_1 \leq x_2 \leq \cdots \leq x_k$.

Lemma 1 *Suppose that Φ is a nonnegative potential function such that*

(1) while the adversary serves a request, Φ cannot increase by more than k times the distance moved by the adversary, and

(2) when we serve a request, Φ decreases by at least the cost we incur in serving the request.

Then $S^* \leq kA^* + \Phi_0$, where Φ_0 is the initial value of Φ.

The proof of this lemma, standard in the study of amortized time analysis, proceeds by a simple summation over the whole sequence of requests. The details are omitted.

Our potential function is $\Phi = \Psi + \Theta$, where

$$\Psi = k \sum_{i=1}^{k} |x_i - s_i|$$

and

$$\Theta = \sum_{i<j} (s_j - s_i),$$

the potential function of [6] specialized to our problem. That this simple potential function (as opposed to our more complicated formulation of the same potential function) indeed works for DOUBLE-COVERAGE was pointed out to us by A. Borodin [3].

Theorem 2 *$S^* \leq kA^* + \Phi_0$, that is,* DOUBLE-COVERAGE *is k-competitive.*

Proof: We will prove the theorem by showing that conditions (1) and (2) from Lemma 1 hold. Lemma 1 then implies that $S^* \leq kA^* + \Phi_0$. ($\Phi_0 = 0$ if all $2k$ servers initially coincide.) We may assume that as a server moves, it does not pass any other servers, ours or the adversary's. Otherwise we can divide the motion of a server into phases, within each one of which the order of the servers remains unchanged. (Of course, at the end of a phase if the moving server "overtakes" another server, we relabel them, if necessary. Doing so leaves Φ unchanged.)

Suppose that x_i moves a distance d within a phase. Then it is clear that Ψ can increase by at most kd, while Θ remains unchanged, and therefore condition (1) holds.

Condition (2) deals only with the second half of the move, after the adversary has served the request. There are two cases: either all of our k servers are on the same side of the request P, or they are not. If all are on one side, say, the right—the other case is similar—our closest server is s_1 and, since the adversary has already served the request, he has a server, say, x_j, at P. Clearly $s_1 \geq x_j \geq x_1$. Moving s_1 to the left d units decreases Ψ by kd and increases Θ by $(k-1)d$, decreasing Φ in total by d. Thus condition (2) holds in this case.

Now, the second case, in which we incur a cost of $2d$. Suppose s_i is our nearest server to P's left, s_{i+1} then being our nearest server to P's right, and let the adversary server on the request site be, say, x_j.

First we analyze the effect of the motion of our servers on Ψ. Only the ith and $(i+1)$st terms can change. If the adversary's server x_j at the request site satisfies $j \leq i$, then the ith term of Ψ will decrease by kd while the $(i+1)$st can increase by at most kd. If $j \geq i+1$, the $(i+1)$st term decreases by kd while the ith can increase by at most kd. In either case, Ψ cannot increase.

Now, we study Θ. The change in Θ due to the movement of our two servers is

$$d[-(k-i)+(i-1)-(i)+(k-(i+1))] = -2d.$$

It follows that (2) holds. ■

3 Cache Problems

We first present a k-competitive algorithm for the weighted-cache problem, a version of the server problem in which each point x is assigned a positive *weight*, $w(x)$. Upon serving a request at x, a server is charged 0 or $w(x)$, according to whether it occupied x or did not occupy x just before serving the request, respectively. This situation is exactly the server problem on the asymmetric metric space in which the distance to x from any other point is $w(x)$. Initially our ith server and the adversary's ith coincide at some point r_i; we assume the r_i's distinct.

Let S^* and A^* be the total cost incurred by us and by the adversary, respectively, over the entire request sequence. We label our servers $s_1, s_2, ..., s_k$ arbitrarily and let S_i always denote the current total of the costs charged to s_i. In this notation, our strategy can be stated as follows.

Algorithm BALANCE: If we have a server at the requested point, we serve the request with that server. Otherwise, we use any server s_a for which $S_a = \min\{S_1, ..., S_k\}$.

We assume without loss of generality that once an occupied point becomes unoccupied (by both us and the adversary) it remains so—a request for it may be replaced by a request for a new point of equal weight. We also assume that there are no requests for points we currently occupy—omitting such a request and the corresponding adversary move leaves our cost unchanged and will, by the triangle inequality, never increase the adversary's cost. We assume, finally, that at the end of the game we and the adversary occupy the same set of points, called the *final* points—subsequent requests for points occupied by unopposed adversary servers would run up our charges and cost the adversary nothing.

Let W_{NF} denote the sum of the weights of the nonfinal points, and let W_F denote the sum of the weights of the final points. Note that $A^* = W_{NF} + W_F$, since each requested point is occupied by the adversary exactly once.

Let T always denote the value of $\min\{S_1, ..., S_k\}$ *just before the most recent service.* We assume its initial value to be 0 and let T^* denote its final value. By $\|s_i\|$ we denote the weight of the point currently occupied by s_i.

Lemma 3 $S_i - \|s_i\| \leq T$, *for* $i = 1, ..., k$.

Proof: Suppose that we served the most recent request with s_j, and choose i arbitrarily. If s_i hasn't served any requests so far, then $S_i = 0$ and hence $S_i - \|s_i\| \leq T$. Otherwise, consider the time just before s_i moved last. At that time, $S_i \leq S_j$, because BALANCE chose to use s_i to serve the request. Currently, S_i is exactly $\|s_i\|$ larger than it was then, and T is at least as large now as S_j was then. Thus, now, $S_i - \|s_i\| \leq T$. ■

Lemma 4 $T^* \leq W_{NF}$.

Proof: Zero initially, T cannot increase until all k of our servers have moved; in fact, T cannot increase from 0 until one has moved twice. It follows that if T increases while s_j occupies y, then at some point in the past s_j served a request at y.

Consider the period of time between time t, just after a server s_j most recently served a request on a nonfinal point y, and time t', just after s_j has vacated y in order to serve some other request. At time t, T is equal to $S' - w(y)$, where S' is the value of S_j at time t. At time t', just after s_j serves a new request, $T = S'$. We conclude that T increases by exactly $w(y)$ during a visit of our server to a nonfinal point y.

We assume that the adversary always moves first to serve a request. Therefore, just after the adversary's move, there is a point that is occupied only by one of our servers. The lemma is proven by allocating the increases in T to such points.

When s_a increases T by serving a request, we do the following. We choose any point y that is occupied, just before s_a serves the request, only by one of our servers, say s_j, and allocate to y the increase in T due to serving the request by s_a. (Perhaps $s_a = s_j$). Previously s_j must have served a request at this nonfinal y. While s_j occupies y, T increases by $w(y)$, and thus the total increase of T allocated to y during this visit of s_j is at most $w(y)$ (it may be less, since some increases in T may be allocated to other points). Since y will never again be requested after s_j leaves y, the total increase of T allocated to y is at most $w(y)$.

All increases of T can be allocated in this way to nonfinal points. Thus the final value T^* of T is the sum of the increases allocated to nonfinal points, and this is at most W_{NF}, by the previous paragraph. ■

We now prove that BALANCE is k-competitive. From [10], we infer that even for the "unweighted" cache problem in which all weights are one (and hence the metric is symmetric), there can be no c_k-competitive algorithm unless $c_k \geq k$. Thus if $c_k < k$ no online algorithm can be c_k-competitive for all instances of the weighted-cache problem.

Theorem 5 *$S^* \leq kA^*$ if no two of our servers coincide at the start.*

Proof: By Lemma 3 we infer that at the end of the game

$$\sum_{i=1}^{k}(S_i - T^*) \;\leq\; \sum_{i=1}^{k} \|s_i\| \;=\; W_F.$$

Using this inequality and Lemma 4 we derive

$$\begin{aligned} S^* &= \sum_{i=1}^{k} S_i \;=\; kT^* + \sum_{i=1}^{k}(S_i - T^*) \\ &\leq kW_{NF} + W_F \;\leq\; k(W_{NF} + W_F) \\ &= kA^*. \end{aligned}$$

■

Note. Bogdan Chlebus [5] proved independently, using a different technique, that BALANCE is k-competitive.

We next consider the unweighted-cache problem, for which several k-competitive algorithms are known: FIFO, LRU, and those of [9] and [13]. Most are not memoryless. We present a new k-competitive memoryless algorithm. Our algorithm, unlike the memoryless algorithm Flush-When-Full of [9], keeps the cache full at all times.

For convenience, we view the metric space for the unweighted-cache problem as the real interval $I = (0, 1]$ by mapping the metric space into I arbitrarily. Initially the k servers occupy any k distinct points. Always, each of our k servers occupies a position given by a real number in I (each such position is the location of some previous request or an initial location). In each time step, a request (a real number $v \in I$) is specified by the adversary. If we have a server at v, we move no servers and pay nothing. Otherwise, we choose one of our k servers and move it to v, incurring *unit* cost. (No two of our servers ever occupy the same point simultaneously. Without loss of generality, nor do two of the adversary's.)

We label our k servers $s_1, s_2, ..., s_k$ arbitrarily, and also use s_i to denote the location of our ith server. We use Algorithm ROTATE.

Algorithm ROTATE: In response to a request at a point $v \in (0, 1]$ unoccupied by any of our servers, if there is an $s_i < v$, choose a maximum such s_i; if no $s_i < v$ exists, choose a maximum s_i. Serve the request at v with server s_i.

Intuitively, our servers move rightward "around" the interval. ROTATE clearly is memoryless. We will show that ROTATE is k-competitive.

For $0 < a, b \leq 1$, we will use $(a, b]$ to mean $\{x | a < x \leq b\}$ if $a \leq b$, and to mean $\{x | a < x \leq 1 \text{ or } 0 < x \leq b\}$ if $a > b$.

Theorem 6 ROTATE *is k-competitive.*

Proof: Label the adversary's k servers $x_1, x_2, ..., x_k$ arbitrarily, using x_i also to denote the location of the adversary's ith server. Let $c_{i,p}$ denote the number of the adversary's servers in $(s_i, x_p]$ and, where π is a permutation of $\{1, 2, ..., k\}$, let $c(\pi) = \sum_{i=1}^{k} c_{i,\pi(i)}$. Our potential function is

$$\Phi = \min_{\pi} c(\pi),$$

the minimum being taken over all permutations π of $1, 2, ..., k$. In other words, Φ is the weight of the minimum weight perfect matching in the bipartite graph with edges $\{s_i, x_p\}$ of weight c_{ip}. We say that π *realizes* Φ if $\Phi = c(\pi)$.

We assume that in response to each request the adversary moves first. We prove the theorem by showing that inequalities (1) and (2) from Lemma 1 hold.

Inequality (1) can be proven as follows. Clearly, we may focus on the case in which the adversary incurs unit cost, by moving x_p (say) to the request site v. We imagine that he moves x_p "rightward" to v, wrapping around the end if necessary. Φ can change only when x_p passes a point occupied only by one of our servers, a point occupied only by one of the adversary's servers, or a point occupied by both. When x_p passes a point containing only one of our servers, Φ increases by at most one. If x_p passes a point occupied only by one of the adversary servers, say x_q, Φ remains unchanged (we may interchange p and q in a permutation π realizing Φ). If x_p passes a point occupied by s_l and x_q, and, say, $\pi(i) = p$, $\pi(j) = q$, then it is not hard to verify that if τ is the same as π except that $\tau(i) = q$, $\tau(j) = p$, then $c(\tau) \leq c(\pi) + 1$. Thus, because we have only k servers, as x_p moves to v the total increase in Φ is at most k.

Before proving (2), let us look at permutations π realizing Φ. Call two intervals $(s_i, x_p]$, $(s_j, x_q]$ *independent* if neither is a subset of the other. If no two s_i's are identical, and no two x_p's are identical, then there is always a permutation π realizing Φ for which pairs of distinct intervals $(s_i, x_{\pi(i)}]$ are independent. For if $(s_i, x_{\pi(i)}]$ is a subset of $(s_j, x_{\pi(j)}]$, let τ be the same permutation as π except that $\tau(j) = \pi(i)$ and $\tau(i) = \pi(j)$; $c(\tau) \leq c(\pi)$. Where the length of $(u, v]$ is defined in the obvious way, the sum of the squares of the lengths of $(i, \tau(i)]$ is strictly less than the sum of the squares of the lengths of $(i, \pi(i)]$, so this process must terminate, with the desired permutation.

Now we prove (2) from Lemma 1. If the request site is occupied by one of our servers, both sides are zero. Otherwise, we incur a cost of one. Suppose that the request site is occupied by x_p, and ROTATE specifies that the request is to be served by s_i. Choose a permutation π realizing Φ for which pairs of distinct intervals are independent.

If $\pi(i) = p$, as s_i moves rightward, $c(\pi)$ decreases by at least one.

Thus we may assume that $\pi(i) \neq p$. If $x_p \notin (s_i, x_{\pi(i)}]$, then, since s_i is the first server of ours to the "left" of x_p, $(s_i, x_{\pi(i)}]$ is a subset of $(s_{\pi^{-1}(p)}, x_p]$, a contradiction. Thus $x_p \in (s_i, x_{\pi(i)}]$ and therefore as s_i moves rightward to serve the request, $c(\pi)$ decreases by at least one. ■

4 Nonexistence Theorems

First we prove that in the asymmetric case in general there can be no competitive algorithm, even if there are only two servers.

Theorem 7 *There is no competitive algorithm for the* 2*-server problem on asymmetric metric spaces.*

Proof: Let K be a positive integer. Where necessary within this proof, we do arithmetic modulo K.

Consider a $3K$-node digraph H_K on vertex set $\{x_0, y_0, z_0, x_1, y_1, z_1, ..., x_{K-1}, y_{K-1}, z_{K-1}\}$ with $4K$ arcs $\{(x_i, z_i), (y_i, z_i), (z_i, x_{i+1}), (z_i, y_{i+1}) | 0 \leq i \leq K-1\}$. Define M_K to be the asymmetric metric space for which $d(x, y)$ is the length of the shortest $x \to y$ path in H_K. Then $d(x_i, y_i) = d(y_i, x_i) = 2K$ while $d(z_i, x_i) = d(z_i, y_i) = 2K - 1$.

If an online algorithm is purportedly c_k-competitive, choose K so that $(2K-1)/4 \geq c_k + 1$ and place all the servers initially at z_0 in M_K. Let a be such that $S^* \leq c_k A^* + a$ for all request sequences whose servers start at z_0.

The adversary's request sequence consists of a sequence of *phases.* At the beginning of the ith phase ($i \geq 1$), the adversary has two servers on z_{i-1}; the online algorithm's servers are anywhere. The adversary starts by requesting x_i and then y_i, and uses two servers to serve the requests. If after the two requests the online algorithm does not have two servers at x_i and y_i, the adversary just alternately requests x_i and y_i, incurring no cost himself, until the online algorithm puts his two servers there (or until the cost of the online algorithm exceeds a more than c times the offline cost to date). Now the adversary requests z_i. If the online algorithm serves the request with the server at x_i, the adversary uses his server at y_i, and *vice versa.* In the first case, the adversary then requests x_i, incurring no cost but costing the online algorithm at least $2K-1$. Next the adversary requests z_i and moves his second server to z_i. (The second case is similar.) This completes the ith phase.

In one phase, the adversary incurs a cost of 4, while the algorithm incurs a cost of at least $2K - 1$. Since $(2K - 1)/4 \geq c_k + 1$, if we run enough phases eventually $S^* > c_k A^* + a$. ■

Our algorithm BALANCE for the weighted-cache problem is not memoryless. Raghavan and Snir [13] presented a randomized memoryless algorithm for the weighted-cache problem. The results of Section 3 show that for the unweighted-cache problem, a memoryless optimal algorithm does indeed exist. The theorem below shows that using either randomization or memory is indeed necessary when arbitrary weights are allowed.

Theorem 8 *There is no memoryless competitive algorithm for the k-server weighted-cache problem.*

Proof: We will show that for each c_k, there is an assignment of positive weights to any finite set of $n \geq k + 1$ points so that no c_k-competitive memoryless algorithm can exist for the associated instance of the weighted-cache problem. Let the vertex set be $\{1, 2, ..., n\}$, $n \geq k + 1$. There are no more than n^k configurations. Let K be positive and assign weight $w(v) = K^v$ to node v. The adversary chooses any strategy that always places the next request on an unoccupied node. Eventually, one configuration C appears a second time (after at most n^k requests, in fact). The adversary now modifies his strategy: from now on, he simply repeats the requests in the cycle between the first and second occurrence of C. The memorylessness of the algorithm ensures that the cycle will be repeated *ad infinitum*. Where m is the highest-numbered node that is requested in the cycle, each iteration through the cycle costs the algorithm at least K^m. The adversary simply uses two servers: one sits permanently on m while the other serves all requests to nodes other than m, thereby incurring a cost of at most $K^{m-1}n^k$ for the cycle. Over an extremely long request sequence, the costs incurred before reaching the cycle can be ignored. Thus, in the limit, the ratio between the cost we incur to that incurred by the adversary is at least $K^m/(K^{m-1}n^k) = K/n^k$. A suitable choice of K makes this ratio exceed c_k. ■

Now we again consider the k-server problem on the line. The algorithm we presented in Section 2 often moved two servers to serve a request. In fact, the following theorem states that no memoryless competitive algorithm on the line always moves at most one server in response to a request. Similar to the previous proof, it uses points K^i on the real line in place of weights K^i. We omit the details.

Theorem 9 *There are no memoryless competitive algorithms for k servers on the line that move only one server at a time.*

5 Offline Problems

First, we study the problem of finding an optimal strategy to serve a sequence of n requests with k servers, if the request sequence is given in advance. We assume that the k servers initially occupy one point, the *origin* (but the algorithm can easily be modified if they don't). When there are n requests, the inputs to our problem are the superdiagonal entries of an $(n+1) \times (n+1)$ matrix, whose $(0, j)$ entry is the distance from the origin to the location of request j, $j = 1, 2, ..., n$, and whose (i, j) entry is the distance from the location of request i to the location of request j, $1 \leq i < j \leq n$.

The dynamic programming algorithm of Manasse *et al.* [10] is especially suited for the case in which the number of requests dramatically exceeds the number m of points in the metric space. Its running time is $O(nm\binom{m}{k})$ and space usage at least $\binom{m}{k}$ for a request sequence of length n in an m-point metric space.

Theorem 10 *There is an $O(kn^2)$-time offline algorithm to find an optimal schedule for k servers to serve a sequence of n requests (whether or not the triangle inequality holds).*

Proof: We reduce the k-server problem—with or without triangle inequality—to the problem of finding a minimum cost flow of maximum value in an acyclic network. If we have k servers $s_1, \ldots, s_k$ and n requests $r_1, \ldots, r_n$, we build this $(2 + k + 2n)$-node acyclic network: the vertex set is $V = \{s, s_1, s_2, ..., s_k, r_1, r'_1, r_2, r'_2, ..., r_n, r'_n, t\}$. Nodes s and t are the source and sink, respectively. Each arc has capacity one. There is an arc of cost 0 from s to each s_i, an arc of cost 0 from each r'_j to t, as well as an arc to t from each s_i, of cost 0. From each s_i, there is an arc to r_j of cost equal to the distance from the origin to the location of r_j. For $i < j$ there is an arc from r'_i to r_j of cost equal to the distance between r_i to r_j. Furthermore, from r_i to r'_i there is an arc of cost $-K$, where K is an extremely large real.

The value of the maximum flow in this network is k. Because all capacities are integral, and because the network is acyclic, we can use minimum-cost augmentation [15] to find an integral min-cost flow of value k in time $O(kn^2)$. An integral $s \to t$ flow of value k can be decomposed into k arc-disjoint $s \to t$ paths, the ith one passing through s_i. Because $-K$ is so small, an integral min-cost flow of value k saturates all of the (r_j, r'_j) arcs, and hence corresponds to an optimal schedule for serving the requests, the ith server serving exactly those requests contained in the $s \to t$ path that passes through s_i. ■

In the next theorem we prove that finding an optimal offline strategy for metric spaces is at least as hard as finding a minimum-weight perfect matching in a complete bipartite graph. This is true even though the metric space distances must satisfy the triangle inequality.

Theorem 11 *If there is an algorithm that computes optimal offline strategies for k-server request sequences of length $2k$ and runs in time $g(k)$, then there is an algorithm that finds minimum weight perfect matchings in $2k$-node complete bipartite graphs in time $g(k) + O(k^2)$.*

Proof: If G is a complete $2k$-node bipartite graph with k boys and k girls, with an edge of weight w_{ij} from boy i to girl j, build a server problem on $2k+1$ points $s, x_1, x_2, ..., x_k, y_1, y_2, ..., y_k$. Initially all k servers occupy s. The length of edge $\{x_i, y_j\}$ is $K + w_{ij}$ where again K is extremely large. The edges $\{s, x_i\}$ are of length K, while all remaining edges are of length $2K$. For K sufficiently large, the triangle inequality holds.

Consider the request sequence $x_1, x_2, ..., x_k, y_1, y_2, ..., y_k$. If K is large enough, an offline strategy which traverses even one edge of length $2K$ is suboptimal. Thus, the first k requests, those to $x_1, ..., x_k$, must be served by k different servers. The same goes for the last k requests. Thus an optimal offline algorithm is one which pairs up the boys and girls so as to minimize the length of the edges traversed, thereby solving the assignment problem. ■

Where $n = 2k$, our algorithm runs in $O(n^3)$ time, the best time known for the assignment problem.

If the metric space is the line, where specifying n request locations requires only n reals, is there a faster algorithm than the one above? Possibly. But there is no linear-time algorithm to find an optimal offline strategy on the line:

Theorem 12 *In the algebraic computation tree model, any algorithm that finds an optimal offline strategy for n requests on the line takes $\Omega(n \log n)$ time.*

Proof: Let $k = n/2$ (for even n) and assume that all k servers are initially at the origin. Given reals $a_1, a_2, ..., a_k$, determining if $a_i \in X = \{3, 3^2, ..., 3^k\}$ for all i is known to require $\Omega(n \log n)$ time in the algebraic computation tree model [1]. Consider the request sequence $3^k, 3^{k-1}, ..., 3, a_1, a_2, ..., a_k$, of length n. We claim that the cost of an optimal offline strategy for this request sequence is $N = \sum_{i=1}^{k} 3^i$ if and only if $\{a_1, ..., a_k\} \subseteq X$. If the latter holds, there is a schedule of cost N: serve the first k requests with k distinct servers, and then incur a cost of 0 for the last k requests. If the first k requests are not served by different servers, then the schedule's cost exceeds N. If the first k requests *are* served by k different servers, the optimal cost can be N only if $\{a_1, ..., a_k\} \subseteq X$. Thus the optimal cost is N if and only if $\{a_1, ..., a_k\} \subseteq X$. ■

Acknowledgements. We would like to thank Piotr Berman, Bogdan Chlebus, Gabor Tardos, Ketan Mulmuley and Janos Simon for many valuable discussions related to our research.

References

[1] M. Ben-Or, Lower bounds for algebraic computation trees, Proc. 15th ACM STOC (1983), 80-86.

[2] A. Borodin, N. Linial, M. Saks, An optimal online algorithm for metrical task systems, Proc. 19th ACM STOC (1987), 373-382.

[3] A. Borodin, personal communication, University of Toronto, Toronto, Ontario.

[4] A. R. Calderbank, E. G. Coffman, L. Flatto, Sequencing problems in two-server systems, Mathematics of Operations Research 10 (1985), 585-598.

[5] B. Chlebus, personal communication, University of California, Riverside, CA.

[6] D. Coppersmith, P. G. Doyle, P. Raghavan, M. Snir, Random walks on weighted graphs, with applications to on-line algorithms, Proc. 22nd ACM STOC (1990).

[7] A. Fiat, R. Karp, M. Luby, L. A. McGeoch, D. Sleator, N. E. Young, Competitive paging algorithms, technical report CMU-CS-88-196, Computer Science Department, Carnegie Mellon University, 1988.

[8] S. Irani, R. Rubinfeld, A competitive 2-server algorithm, manuscript, UC Berkeley.

[9] A. Karlin, M. Manasse, L. Rudolph, D. Sleator, Competitive snoopy caching, Algorithmica 3 (1988), 79-119.

[10] M. Manasse, L. A. McGeoch, D. Sleator, Competitive algorithms for online problems, Proc. 20th ACM STOC (1988), 322-333.

[11] M. Manasse, L. A. McGeoch, D. Sleator, Competitive algorithms for server problems, Tech. Rep. CMU-CS-88-197, Carnegie-Mellon University,

[12] L. McGeoch, D. Sleator, A strongly competitive randomized paging algorithm, to appear in Algorithmica.

[13] P. Raghavan, M. Snir, Memory versus randomization in online algorithms, Proc. Sixteenth ICALP (1989), to appear in Springer-Verlag Lecture Notes in Computer Science 372.

[14] D. Sleator, R. E. Tarjan, Amortized efficiency of list update and paging rules, Comm. ACM 28 (1985), 202-208.

[15] R. Tarjan, Data Structures and Network Algorithms, SIAM, Philadelphia, 1983, 109-111.